CHAPTER 3

Building a Simple Application

In This Chapter

Understanding the desktop application categories

Determining which desktop application type will work best for your needs

Creating simple console, dialog-based, SDI, and MDI applications

Windows NT offers an extremely flexible programming environment that you can use to create a variety of application types. One of the more common applications that you can create is the desktop application. A desktop application is designed to interact directly with the user in some way. We've all used desktop applications in the form of a word processor or the IDE (integrated development environment) of a programming language product. Contrast a desktop application to something like a service that operates fully in the background without any user interaction at all.

Most programmers would instantly know whether they wanted to create a desktop application. However, programming products like Visual C++ enable you to create a variety of desktop application types. Each type of desktop application is best at performing a certain type of user-interaction task. It's important to choose the right kind of desktop application project from the outset. We'll start with learning about the various application types that you can create with Visual C++ and which ones work best in certain situations.

Once you have a programming project type in mind, you can create an application shell, then fill it out to create a fully implemented application. There are actually six basic desktop application types, but we'll only discuss four of them in this chapter. Internet variations of the desktop applications will appear later in the book. For the most part, other C++ application types have replaced the Win32 application type, so we won't discuss it in the book. The last four sections of this chapter are going to show you how to create the four basic desktop applications that Visual C++ supports.

There's one reason that you may want to use the Win32 project; it doesn't require any MFC support. This could be a plus on very small projects where application size and ease of distribution are more important than development speed. However, in most cases, you'll still want to bypass the Win32 project type for the newer project types listed in Table 3-1.

Desktop Application Categories

Visual C++ is quite capable of creating any application you can imagine (and perhaps a few that you can't). However, there are six application types that exemplify desktop applications as a whole, and that's what we'll concentrate on first. Table 3-1 contains a list of the typical application types and what you'll normally use them for.

Table 3-1
Basic Application Types for Visual C++ 6.0

Application Type	*Description*
Console	Represents those situations in which you really need to maintain some type of compatibility with legacy systems or you don't need a full-fledged interface for the user to work with. Console applications look like DOS applications, but require you to use Windows code.
Win32	This is one of the oldest application types that you can create for Windows, and represents the lowest common denominator when it comes to desktop-application discussions. The three starting points for this project type are an empty shell, an application shell that includes a simple WinMain function, and a Hello World(type application that provides you with a simple menu and About Box. For the most part, the other desktop application types we'll discuss in this chapter have superceded this project.
Dialog-based	Normally reserved for utilities or an application that's too small to require a complete menuing system. The standard dialog-based applications rely on one or more dialog boxes to contain all of the data-entry controls and pushbuttons that are required to perform a specific task.

Application Type	*Description*
Single-document interface (SDI)	Representative of simple applications that work with their own data like note takers or small database front ends. These applications also require a menuing system of some type. The main thing to remember about SDI applications is that they normally work with a single document at a time, even if the user has more than one document loaded at the time.
Multiple-document Interface (MDI)	Includes full-fledged applications like word processors and spreadsheets. When you think about it, MDI applications represent that fringe area of C++ programming where you need to weigh the flexibility of C++ against the development speed offered by RAD programming environments like Visual Basic. MDI applications are normally designed to work with more than one document at a time. This functionality allows you to do things like compare the contents of one word-processed document against another.
HTML-based	Applications that work with data of some type (like SDI or MDI applications), but with an Internet twist. HTML-based applications are new to Visual C++ 6.0. Instead of a standard editor, your user will see what amounts to a Web-browser front end. Normally, this Web-browser front end will view HTML code rather than the strict text or graphics used by most applications.

Remember that we're talking about standard desktop applications in this chapter. Visual C++ is capable of creating all kinds of different code, including a new HTML application type that enables you to work with the Internet from your desktop. You can use Visual C++ to create DLLs, ActiveX controls, ISAPI extensions, device drivers, background executing programs like screen savers, and even extensions to Visual C++ itself. We're only talking about general applications in this chapter, but we'll cover many of these other possibilities as the book progresses.

Deciding on an Application Type

There's a multitude of application types that you can create with Visual C++; imagination is really the only limiting factor. However, because of the way that Visual C++ sets up projects, you can divide the number of application types into the six categories discussed in the previous section (four of which will be discussed in this section of the book).

The Win32 application type is provided for compatibility reasons only; you really shouldn't use it for new project development (unless you really want to type a lot of code by hand). The dialog-based, SDI and MDI application types represent refinements of the original Win32 application, and it's much easier to develop a new application using them. We won't discuss the HTML-based application in this section because the HTML application is really a new way to view data using either an SDI or an MDI application as a starting point.

Deciding between the four major categories is the first step in designing any desktop application. In this section, we'll discuss the various application categories and when you would use them. Some situations exist in which an application could conceivably overlap two categories. In those cases, I'll also discuss the potential tradeoffs you'll need to make when choosing one category over another.

Console

I made some statements previously that gave you an overall sense of what a console application is, but they were too sweeping to really tell you what a console application is all about. A console application has a DOS window look rather than the more familiar Windows-style appearance. It uses a monospaced font, just like you would see in a DOS window, and you can use standard C functions for output like `printf()` and `scanf()`. In other words, a console application can use the application logic that you used in older DOS applications, should you still have any around to convert. However, internally, the console application is still a Windows application — it won't run under DOS. Figure 3-1 shows a typical example of a console application (we'll actually build this one later in the chapter).

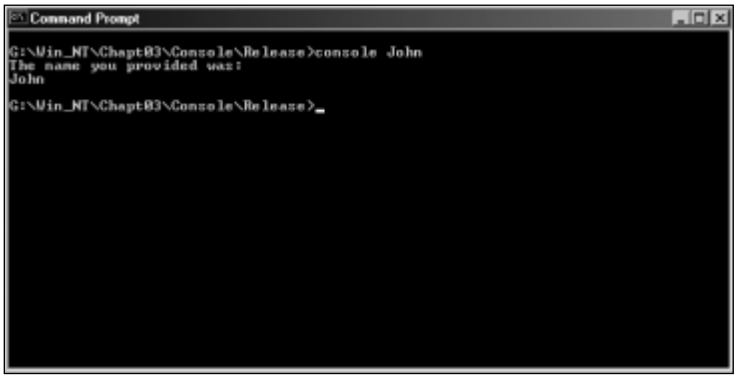

Figure 3-1: Console applications use what appear to be a DOS window and a monospaced font.

So, why would you want to create a crossbreed monstrosity like this? Say you have some old legacy code that works just fine, but it's built into a DOS application. You want to move the program to Windows, but don't have the time to convert everything over to standard Windows calls. A console application provides a sort of middle ground. You can use that old code (at least part of it), but still execute the program within Windows.

There's another benefit to working with console applications. If you move your application from DOS to Windows, it's going to look different. There isn't any way around it; from a user perspective you may just as well introduce a totally different application. You're going to have to plan on retraining everyone and everything. On the other hand, a console application will look enough like the DOS application everyone is used to using that you probably won't have much, if any, retraining to do. The only thing you'll need to worry about is getting everyone used to running the application in a different way.

Another way that programmers use console applications is as a way of writing very small, easy-to-distribute applications that don't require much memory. You can build a functional console application that consumes as little as 28KB and requires only Kernel.DLL to execute. In other words, the dependency list for this application type is almost nonexistent. Console applications are perfect for very small tasks that require little user intervention, although you'd still want the user to see the final result of whatever task the application performed.

Don't get the idea, though, that a console application is the ideal setup for every DOS application you want to move to Windows. The bottom line is that you're going to create a new application either way. The only difference is the amount of code reuse and, hence, the amount of time it'll take to create the new application. In fact, when you think about it, if you decide to go the console application route, you'll eventually write two applications. Most companies find that they eventually have to bear the cost of converting their DOS applications to Windows anyway.

Of course, you also have to ask yourself about the advisability of writing a console application when the DOS application works just fine within a DOS window. Sure, a console application can use some Windows services, but you probably have all the services you need already built into the DOS application. (You'll have access to some MFC functions that could augment your application and make it easier to use.) The more you deviate from your original DOS application to add Windows-style bells and whistles, the less viable a console application becomes.

It's safe to assume that you'll be able to move the "business logic" of your DOS application to Windows using a console application. You may also be able to move some of the display and printing elements. However, it's never safe to assume that you'll be able to maintain one set of code for both DOS and Windows by using a console application—the two environments are different enough that you'll always have to make changes to the code in the move from DOS to Windows.

Dialog-based

Many people associate dialogs with configuration screens, About boxes, and other adjuncts to a full-fledged application. Dialog-based applications have a place too. They're extremely useful for utility-type applications in which you need to display a fairly minimal amount of data and require minimal user input. You can also use them for an informational display—like a dialog box that presents the amount of free memory the computer has left. Finally, you can use them for external configuration dialogs for applications, services, or the operating system as a whole. Figure 3-2 shows an example of a dialog-based application that we'll build later in the chapter.

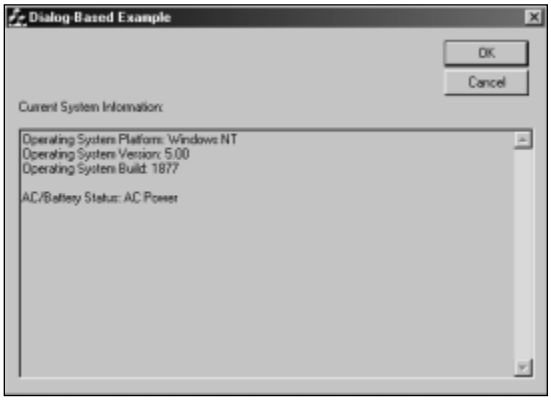

Figure 3-2: Dialog-based applications are most useful for utility, configuration, and informational applications.

When deciding whether to build a dialog-based or window-based application, think utility. If your application fits into the utility category, it's probably a good candidate for a dialog-based interface. On the other hand, if you're thinking about adding a lot of features or allowing the user to interact with the application heavily, you may want to look at a window-based (SDI or MDI) interface. Make sure you take future expansion into account when making your decision—a bad choice today could cost you a lot in rework time tomorrow.

So, what makes a dialog-based application better than a full-fledged window-based application? One of the more important factors is size. You can create two versions of the same application, one that uses a dialog front end and another that depends on a window. The dialog version will be smaller every time. In addition to conserving resources, you may find that the dialog version loads faster. A dialog-based application is simply more efficient than its window-based counterpart.

You may find that building a dialog-based application is faster as well. Dialog-based applications are meant to be small and efficient. If you find that you're adding a lot of bells and whistles to this kind of application, perhaps you've used the wrong kind of application. Dialog-based applications normally eschew menus and other paraphernalia that a window-based application requires to provide a user-friendly front end. Fewer bells and whistles spell reduced development and debugging time for the programmer. Obviously, anything that speeds the programmer along is worth some consideration.

Dialog-based applications don't have to suffer from lack of essential features either. For example, you can create a perfectly acceptable OLE server out of a dialog-based application. The wizards in Visual C++ will even help you out in this area, so adding OLE support requires a minimal effort on your part.

The only real problem with dialog-based applications is that some programmers feel they can stuff one to the point of breaking. I've seen some dialog-based applications that are so filled with gizmos that you really can't tell what they're supposed to do. While a dialog-based application may look a little more cramped than its window-based counterpart, you shouldn't have it so crammed that no one can use it.

You can reduce clutter on a dialog-based application by using tabbed pages, just like a property dialog used by Windows for configuration purposes.

Single-document Interface (SDI) Applications

An SDI application is an application like NotePad or Microsoft Paint. It's designed to handle one document at a time, which reduces programming complexity and the amount of resources required to run the application. You'd use this kind of windowed application for something that doesn't require multiple document access, like a text editor or perhaps a small graphics editor. An SDI application enables users to interact fully with the document that you want them to create, but it's usually less robust than an application designed to work with multiple documents. In addition, an SDI application usually has a minimum of one less menu than a multiple-document application would — the Window menu that is used to select the document you want to edit.

There isn't anything to stop a programmer from trying to get the standard SDI application to work more like an MDI application. For example, you could allow the user to load more than one document at a time. The limitation of SDI is that you can't display all of the loaded documents at the same time. (If a programmer were to allow the user to load more than one document at a time, a Windows menu would be essential since there would be no other way to select a different document.) The chief difference between SDI and MDI applications is that an SDI application only displays one document at a time.

Like the dialog-based application, your SDI application can act as an OLE server. In fact, you'll find that it can also act as an OLE client, even though very few programmers add this capability to their program. Figure 3-3 shows an example of an SDI application we'll create later in the chapter. Notice that it's acting as an OLE client in this case.

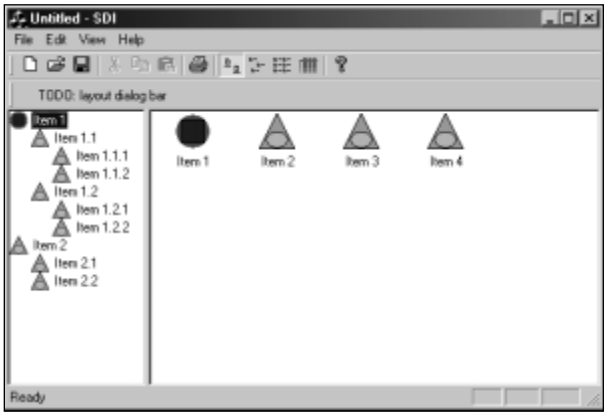

Figure 3-3: SDI applications always handle one document at a time, even if the application is designed to load more.

You can convert your SDI application to a simple Web browser by using the CHtmlView class as your base view class. In fact, the SDI application type allows you to create a browser lookalike that can perform tasks other than browse the Web. We'll look at the requirements for building Internet applications later in the book.

Unfortunately, an SDI window-based application can suffer from the same problem as a dialog-based application — too much complexity. I still remember problems I had trying to use older versions of Corel Draw. Every time I wanted to look at a drawing, I had to close the currently open document before doing so. This limitation made Corel Draw a little harder to use than it needed to be. For example, I wasted a lot of time trying to compare one drawing against another. (Fortunately, Corel Systems has corrected this oversight in current versions of Corel Draw by allowing the user to open more than one document at a time; even though only one document can be displayed at any given time.)

The SDI, window-based application works surprisingly well when it comes to database management systems. The reason is fairly simple: Very few (if any) of your users will need to open more than one database at a time. Even if they do, the rules for working with databases would make it less desirable to allow the user to access more than one database at a time. You'll normally want to control all access to the various database elements programmatically and display the results to the user.

Multiple-document Interface (MDI)

Now we're going to talk about the MDI application. You'd use this kind of window-based application to create something like a word processor or spreadsheet application. For example, both Microsoft Word and Microsoft Excel are examples of MDI applications. If you think about it for a second, a text editor has a limited appeal simply because it can only open one document at a time. People need to be able to compare one document to another; that's why MDI applications are not only nice, but required in so many situations. Figure 3-4 shows an example of an MDI application in action. Notice that there's more than one document in view at a time. We'll build this application later in the chapter.

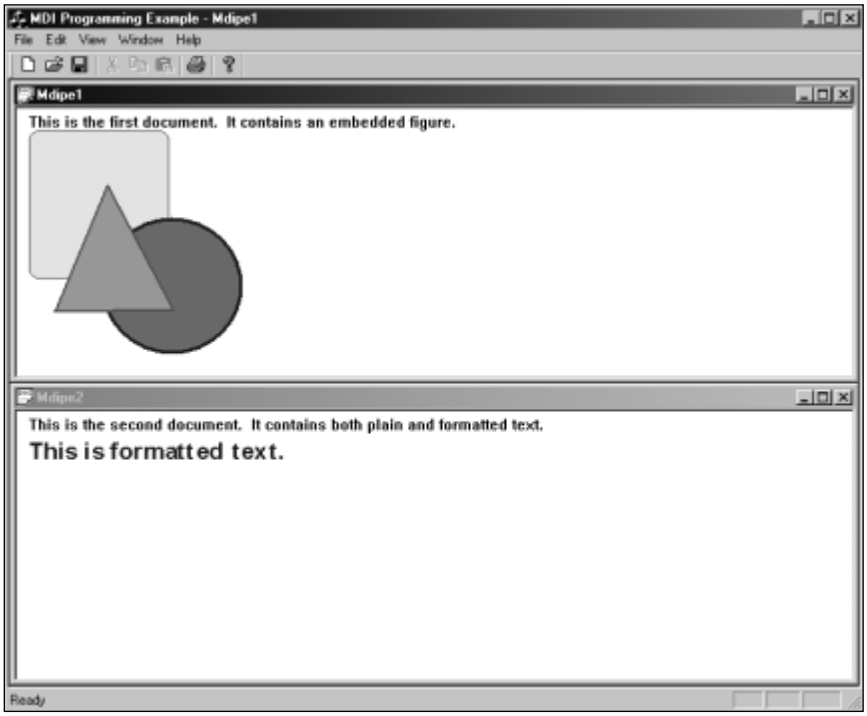
MDI Programming Example - Mdipe1
File Edit View Window Help
Mdipe1
This is the first document. It contains an embedded figure.
Mdipe2
This is the second document. It contains both plain and formatted text.
This is formatted text.
Ready

Figure 3-4: MDI applications permit the greatest document flexibility by allowing you to view more than one document at a time, but they tend to be complex to write.

MDI applications also tend to be feature rich. (You can still go overboard; just look at all the people complaining about the bloated feature sets of major products produced by vendors today.) A text editor may provide a very simple find function and not offer any means for replacing occurrences of specified text. A full-fledged word processor provides both search and replace as standard features.

You can convert your MDI application to a simple Web browser by using the CHtmlView as your base view class. The advantage of using an MDI application over an SDI application is that you can track multiple Web sites for comparison purposes. However, this added functionality comes at a price—complexity. Writing an application that can track multiple Web sites at the same time is no small undertaking. We'll look at the requirements for building Internet applications later in the book.

The failings of an MDI application begin with the fact that it can handle multiple documents. The capability of handling more than one document at a time means a lot of additional programming. Not only do you have to keep track of all the open documents but you have the Window menu to manage and special program features like split screen to consider as well. You'll also need to decide whether the user will be allowed to display more than one document at once. Things like minimizing one document while keeping another maximized require additional code as well. To sum it up, you'll need to be prepared for some major programming time before you even start an MDI application.

Of course, MDI applications have plenty of other disadvantages as well. For example, if you've ever tried to use Word for an OLE server, you know all about the frustration of waiting for this behemoth application to open every time you click on a link in another application. You've probably experienced the serious consequences of running out of memory as well. Until recently, every time you wanted to use OLE, you had to have enough memory to run both applications (the client and the server). Fortunately, Microsoft has reduced this requirement somewhat by allowing the server to take over the client's window; now the server only has to worry about working with the document itself. The client window provides a framework for the server's menus and toolbar, so there isn't any extra memory wasted.

A Simple Console Application

As previously mentioned, console applications rely on a DOS-type window to display information. You can use a variety of techniques for creating this project, including the use of MFC. The most common use of a console application is to allow your old DOS code to live anew under Windows. However, you can use console applications for a variety of other purposes as well, including executing small batch-type routines that require little or no user intervention.

In this section, we'll create a simple console application that demonstrates some important features some of you may not know exist. For example, our sample application will include a class definition. The following procedure will get you started by showing you how to create console applications in general.

Try This

1. Open Visual C++ if you haven't done so already.
2. Use the File | New command to display a New dialog box like the one shown in Figure 3-5. Notice that I've already chosen the Projects tab and highlighted the project type that we'll use.

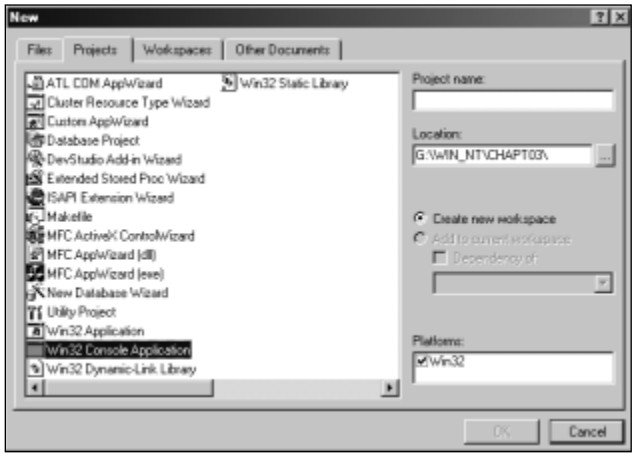

Figure 3-5: The Projects tab of the New dialog allows you to choose a project type to create.

3. Once you choose Win32 Console Application, type a name for your program in the Project Name field. The sample program uses the name **Console**. You may need to change the contents of the Location field as well. Just click the browse button next to the field and you'll see a Choose Directory dialog box, in which you can choose a destination directory for your application.
4. Click OK. You'll see the Win32 Console Application — Step 1 of 1 dialog shown in Figure 3-6. Notice that you have a choice of several application types to get you started. This is a new feature of Visual C++ 6.0. Previous versions of Visual C++ would simply create an empty project. The empty project is the one that you could create with previous versions of Visual C++ 6.0. Choose the "A simple project" option if you want a project that includes both stdafx.H and an empty Main function. The "A Hello World application" adds some sample code to your application and automatically records the application options. The final option, "An application that supports MFC", allows you to create more complex applications that include standard MFC classes like CString.

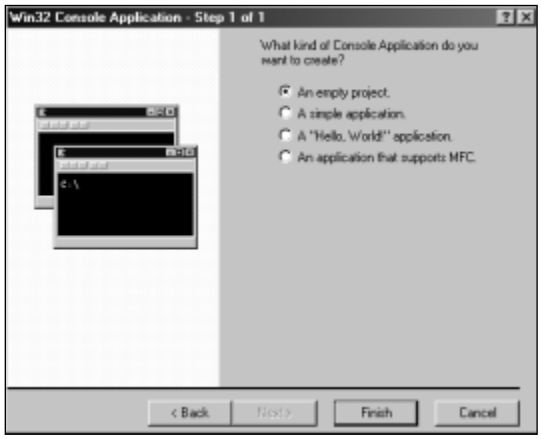
Win32 Console Application - Step 1 of 1
What kind of Console Application do you want to create?
An empty project.
A simple application.
A "Hello, World!" application.
An application that supports MFC.
< Back
Next >
Finish
Cancel

Figure 3-6: There are four flavors of Win32 Console Application, including the empty application used by previous versions of Visual C++.

5. Choose an application type and click OK. The example uses the "A simple application" option.

6. Click Finish and you'll see a New Project Information dialog. Figure 3-7 shows the New Project Information dialog box for the example project. Notice that the New Project Information dialog box contains a list of all Wizard options you selected. This is your last opportunity to ensure you have your application shell set up correctly before you begin to code, so it makes sense to pay particular attention to this dialog.

Figure 3-7: The New Project Information dialog box enables you to check your application settings before starting to code.

7. Click OK. Visual C++ will create the application shell for you.

Gaining access to MFC functions from within a console application is easy; simply include AFXCOLL.H at the very beginning of the code. This file contains all the defines and class definitions you'll need to implement a limited number of MFC calls within your console application. However, you can't use every MFC class. For example, you can't create a CDialog object and then actually expect to use it. A good rule of thumb to go by when deciding which MFC classes to use is to see if the class works with graphical elements. If it does, there's no chance of you using it within a console application. In addition, you'll find that certain system calls are out of reach and that you'll have to exercise care when it comes to security and disk access. If in doubt, stick with the calls listed in AFXCOLL.H (and any associated header files like AFX.H) to the exclusion of everything else. You can safely use all of the calls within AFXCOLL.H in any console application.

Now that we have a program shell in place, let's add some code so that we can see a console application in action. In this example, we'll do something fairly simple—display the user's name. Listing 3-1 provides the source code for this application. You can see the output of this application by looking at Figure 3-1.

CD-ROM Location: Chap03\Console
Program Name: Console.EXE
Source Code Modules in Text: Console.CPP

Listing 3-1: **Simple Console Application Source Code**

```
// Console.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "stdio.h"

class CDispText
{
public:

   // Displays the text.
   void DoIt(char* string);
};

void CDispText::DoIt(char* string)
{
   // Display the text.
   fprintf(stdout, "The name you provided was:\n%s\n", string);
}

int main(int argc, char* argv[])
{
```

Continued

Listing 3-1: *(continued)*

```
    char*          cName;     // Name of person typed at command line.
   CDispText   oText;   // A text display object.

   // See if we have enough command line arguments.
   if (argc != 2)
   {
        // If not, display an error message.
        fprintf(stderr, "Type the program name followed by your name.\n");
        return 1;
   }

   // Get the command line arguments.
   cName = argv[1];

   // Display the test.
   oText.DoIt(cName);

   return 0;
}
```

Even though no one would actually create an application like this, it does have several interesting features. Program execution begins as it does for most C/C++ applications of the DOS era, at main( ) function. This function begins by seeing if there are enough arguments provided at the command line. If not, the program exits with an error code of 1 and an error message. Otherwise, the main( ) function places the argument in a string and passes it along to our CDispText class (the actual object name is oText in this case).

The CDispText class includes only one method, DoIt( ). This method doesn't do very much; it just displays a string followed by the name passed at the command line by the user. The important thing to remember is that a console application does give you the ability to use classes. It doesn't matter if those classes are MFC-based or of your own creation.

Console.EXE can be added to your start menu. You can add command-line parameters to it just as you would for any other application. Alternative execution methods include using batch files or direct execution at the DOS prompt (shown in Figure 3-1). Here's an example of a batch file you could use to execute your console application.

```
@ECHO OFF
CONSOLE JOHN
IF ERRORLEVEL==1 ECHO IT'S BAD
@ECHO ON
```

One of the biggest benefits of this particular application, when compared to other Windows applications, is size. Our example program requires 28KB of disk space and absolutely nothing in the way of support files.

A Simple Dialog-based Application

Dialog-based applications are good for a lot of small tasks. In most cases, they consist of a single dialog box that includes all of the controls required to perform a specific task. Good examples of dialog-based applications include utilities and monitors.

In this section, we'll create a simple dialog-based application that demonstrates a typical task for this type of application. We'll grab some system information like the operating system build number and display it in an Edit Box control. The following procedure will get you started creating any dialog-based application.

Try This

1. Open Visual C++ if you haven't done so already.
2. Use the File | New command to display a New dialog box. Click the Projects tab so that you see a dialog box like the one shown in Figure 3-5.
3. Highlight the MFC AppWizard (exe) icon. This is the icon you'll use to start most of the desktop-level applications in this book.
4. Type a name for your program in the Project Name field. The sample program uses the name **Dialog**. You may need to change the contents of the Location field as well. Just click the browse button next to the field and you'll see a Choose Directory dialog box in which you can choose a destination directory for your application.
5. Click OK. You'll see the MFC AppWizard Step 1 dialog box shown in Figure 3-8. Notice that I've already chosen the Dialog-based option and that the MFC AppWizard displays a project of this type on the left side of the dialog box.

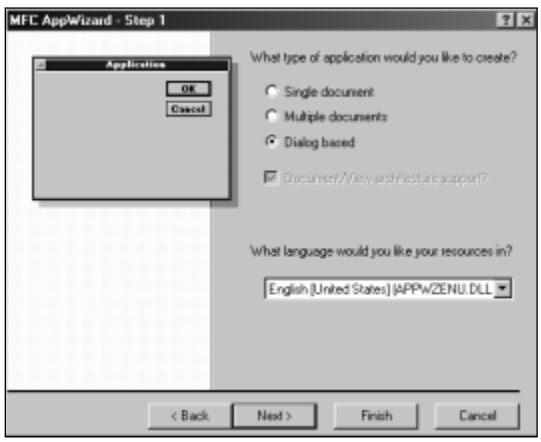

Figure 3-8: The MFC AppWizard Step 1 dialog box gives you a choice of three application types to create.

6. Choose the Dialog-based option, then click Next. You'll see the MFC AppWizard Step 2 of 4 dialog box shown in Figure 3-9. This dialog box allows you to set many of the default features of your dialog box, including whether it has an About Box and a Help button. The Automation option allows you to add support for manipulating objects contained in another application, or exposing objects in your program for another application to manipulate. If you were creating some type of configuration utility, you'd probably want to check this option to make it easier to configure the application's settings. Since all of the controls you'll use for your dialog box are ActiveX controls, you'll want to keep the ActiveX Controls option checked. Check the Windows Sockets if you want your dialog-based application to communicate with the Internet or any other remote location that relies on TCP/IP. Finally, you'll need to type what you'd like to see displayed on the title bar of your application.

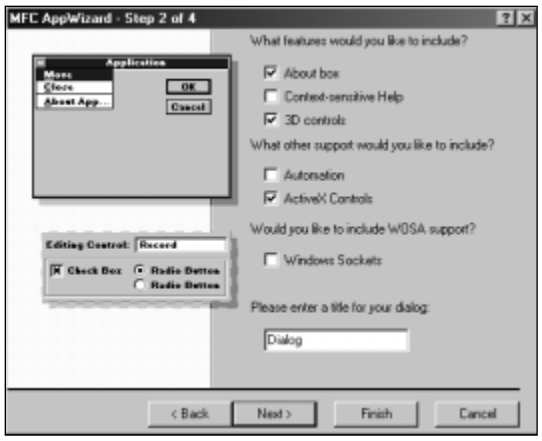

Figure 3-9: You'll want to check your dialog-based application settings to ensure that all the features you want appear in the application shell.

7. Choose one or more of the options in the MFC AppWizard Step 2 of 4 dialog box. The sample application uses the default settings. You will want to change the dialog title bar text to "Dialog-Based Example."

8. Click Next. You'll see the MFC AppWizard Step 3 of 4 dialog box shown in Figure 3-10. This is where you'll determine the characteristics of the code the AppWizard generates for you. First, you need to decide whether to add comments to the source that the wizard generates automatically for you. Always generate the comments because there isn't a good reason not to do so, and trying to figure out some MFC code without comments can be difficult. Second, you have a choice between statically linking MFC to your application or creating a dynamic link. This particular choice can impact your application in a big way. Statically linked applications are huge because all of the required MFC library routines are included with the application. On the other hand, if dynamic linking is used, your application relies on the MFC library being available externally. This option can lead to problems when the user's machine has an older version of the libraries than what you used to develop the application.

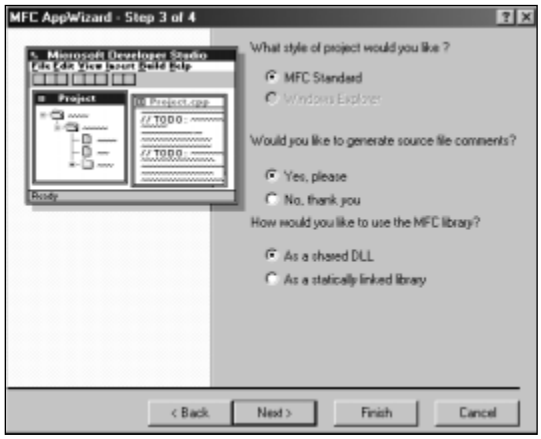
MFC AppWizard - Step 3 of 4
What style of project would you like ?
MFC Standard
Windows Explorer
Would you like to generate source file comments?
Yes, please
No, thank you
How would you like to use the MFC library?
As a shared DLL
As a statically linked library
< Back
Next >
Finish
Cancel

Figure 3-10: This dialog allows you to include source comments and MFC linking type.

9. Choose the application characteristics that you want. The example application uses the default settings of generating the source comments and dynamically linking MFC. In most cases, you should choose these options as well.

10. Click Finish. You'll see the New Project Information dialog shown in Figure 3-11. Remember that this is your last opportunity to check the program generation options before you begin coding. Always make sure that the application skeleton will start out with the features you need to develop the application.

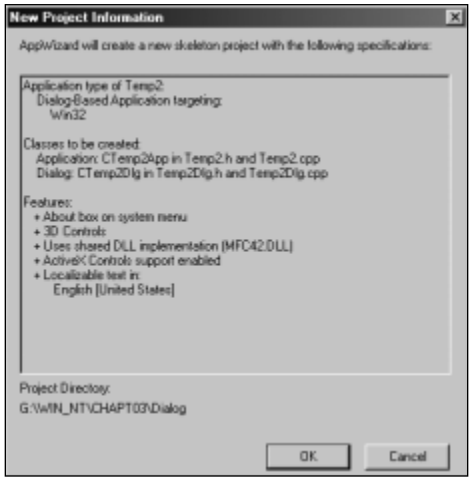

Figure 3-11: The New Project Information dialog shows that we're creating a dialog-based application with standard features.

11. Click OK. Visual C++ will generate your application. However, instead of seeing the Code Window right away, you'll see a Dialog Editing Window.

The first step in fleshing out our sample application is to add some controls to the dialog box. Two of the controls are relatively simple to add. We need a Static Text control to display a heading for the information our dialog will provide and an Edit Box to display the information. Right-click the Static Text control and choose Properties from the context menu. Type **"Current System Information:"** in the Caption field located on the General tab. Click the close box. Right-click the Edit Box control and choose Properties from the context menu. Type **"IDC_DISP_PROP"** in the ID field of the General tab. You'll also need to check the Multiline, Vertical Scroll, and Read-only options on the Styles tab.

We'll also need a copy of the Microsoft SysInfo ActiveX control. To add it, right-click the dialog and choose Insert ActiveX Control from the context menu. You'll see an Insert ActiveX control dialog. Find the Microsoft SysInfo entry, highlight it, then click OK. Figure 3-12 shows what your dialog should look like at this point (you'll need to arrange your controls to match mine).

Figure 3-12: Our dialog box will contain the ActiveX controls required to capture and display the system information settings.

Adding controls to a dialog box doesn't make them accessible. We have to create member variables that will enable us to access the two most important controls on this dialog box: IDC_DISP_PROP and IDC_SYSINFO1. Ctrl-double-click IDC_DISP_PROP. You'll see an Add Member Variable dialog box. Type **m_dispProp** in the Member Variable Name field and choose Control in the Category field. Your dialog should look like the one shown in Figure 3-13. Click OK to create the member variable.

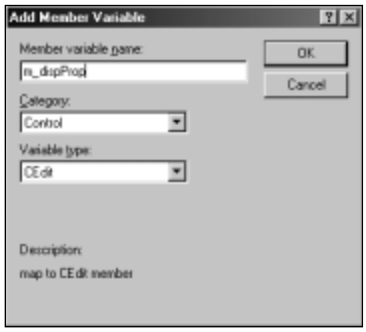

Figure 3-13: The Add Member Variable dialog allows you to add variables to your application that can access the controls on a dialog.

You'll also need to create a member variable for IDC_SYSINFO1. However, when you Ctrl+Double-click on the IDC_SYSINFO1 control, Visual C++ tells you that you haven't created a wrapper class for it yet. Click OK, Visual C++ displays a Confirm Classes dialog like the one shown in Figure 3-14. This class will allow you to access the features of the Microsoft SysInfo ActiveX control on the dialog. Click OK and Visual C++ will create the wrapper class for you. At this point, you'll see the Add Member Variable dialog shown in Figure 3-13. Type **m_sysInfo** in the Member Variable Name field, then click OK. Visual C++ will add the required member variable to our application.

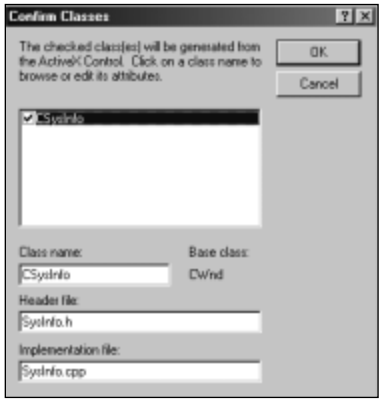

Figure 3-14: The Confirm Classes dialog box allows the programmer to add a wrapper class for any ActiveX controls used in the application.

It's finally time to add some code. In this case, we only need to add code to the OnInitDialog( ) method of the DialogDlg.CPP file. Placing the code here means that the dialog will contain the system information data that we want as soon as the application starts. Listing 3-2 lists the code you'll need to add to the example to make it work. Figure 3-2 shows the application in action.

CD-ROM Location: Chap03\Dialog
Program Name: Dialog.EXE
Source Code Modules in Text: DialogDlg.CPP

Listing 3-2: **Simple Console Application Source Code**

```
BOOL CDialogDlg::OnInitDialog()
{
   int          iOSPlatform;
   double       dOSVersion;
   int          iOSBuild;
   int          iACStat;
   int          iDecimal;
   int          iSign;
   CString      sOSPlatform;
   CString      sOSVersion;
   LPTSTR       sOSBuild = "";
   CString      sACStat;
   CString      sDispData;

   CDialog::OnInitDialog();

   // Add "About..." menu item to system menu.

   // IDM_ABOUTBOX must be in the system command range.
   ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
   ASSERT(IDM_ABOUTBOX < 0xF000);

   CMenu* pSysMenu = GetSystemMenu(FALSE);
   if (pSysMenu != NULL)
   {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
             pSysMenu->AppendMenu(MF_SEPARATOR);
             pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
   }

   // Set the icon for this dialog.  The framework does this automatically
   //  when the application's main window is not a dialog
   SetIcon(m_hIcon, TRUE);           // Set big icon
   SetIcon(m_hIcon, FALSE);          // Set small icon

   // Get the operating system platform.
   iOSPlatform     = m_sysInfo.GetOSPlatform();
   switch(iOSPlatform)
```

```
    {
    case 0:
         sOSPlatform = "Win32S";
         break;
    case 1:
         sOSPlatform = "Windows 95";
         break;
    case 2:
         sOSPlatform = "Windows NT";
         break;
    }

    // Get the operating system version.
    dOSVersion = m_sysInfo.GetOSVersion();
    sOSVersion = _fcvt(dOSVersion, 2, &iDecimal, &iSign);
    sOSVersion = sOSVersion.Left(iDecimal) + "." +
         sOSVersion.Right(sOSVersion.GetLength() - iDecimal);

    // Get the operating system build.
    iOSBuild = m_sysInfo.GetOSBuild();
    _itoa(iOSBuild, sOSBuild, 10);

    // Get the AC/Battery status.
    iACStat = m_sysInfo.GetACStatus();
    switch(iACStat)
    {
    case 0:
         sACStat = "Battery";
         break;
    case 1:
         sACStat = "AC Power";
         break;
    case 255:
         sACStat = "Unknown";
         break;
    }

    // Build a display string.
    sDispData = "Operating System Platform: " + sOSPlatform +
         "\r\nOperating System Version: " + sOSVersion +
         "\r\nOperating System Build: " + sOSBuild +
         "\r\n\r\nAC/Battery Status: " + sACStat;

    // Display the information.
    m_dispProp.SetWindowText(sDispData);

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Some of this code is provided by the MFC AppWizard to display the dialog box. The first thing you'll notice in the example code is the declaration of both integer (or double) and CString versions of each system information statistics. The Microsoft SysInfo control returns all of its statistics as numbers, so we need to convert them to text. The actual conversion process varies by statistic.

Let's look at the system platform statistic. As shown in the code, the SysInfo control returns a value of 0, 1, or 2, depending on the current platform. All we need to do is use a switch statement to convert the number to a text equivalent. This same process works for the AC/Battery status statistic as well. However, notice that, in this case, a return value of 255 means that the system is unsure about battery or AC power status. There aren't many cases in which Windows NT will actually return this value, though older versions of Windows might.

The operating system version statistic is returned as a floating-point number. The number before the decimal point is the major version number; the number behind the decimal point is the minor version number. The example uses _fcvt( ) function to convert the floating-point number to a string. Unfortunately, this string doesn't have a decimal point in it. That's where the CString Right( ) and Left( ) methods come into play. Since _fcvt( ) returns the position that the decimal point is supposed to occupy, it's no problem to add it into the string as shown in the code.

The operating system build is returned as an integer. All you need to do is use the _itoa( ) function to convert it to a string. In this case, a LPTSTR works better than a CString. You'll want to be sure to initialize your LPTSTR before using it.

The last thing we need to do is build a string containing all of our statistics and place it in the Edit Box control. Notice that the individual lines of the string are separated by an \r\n escape sequence. The order in which you place the escape sequence is important because the edit box only interprets the \r\n sequence as a new line. Everything else will appear as a control character within the Edit Box control.

A Simple SDI Application

You'll normally see SDI applications used in places where there isn't much of a need to compare one file to another. However, this isn't the only place you'll see SDI applications used. There's a wealth of specialty applications that rely on SDI as well. For example, Windows Explorer only displays one "document" at a time using what's known as the Explorer view. In other words, the document, in this case, represents a framework used to display both a hierarchy of your drive layout and the individual icons in the selected folder.

This section of the chapter is going to look at one of the newer features that Visual C++ allows you to add to your application, the Explorer interface. We'll use it to

build a very simple display that uses both the CTreeView and CListView classes. While this simple example won't do anything particularly wonderful, you can use it as a starting point for your own application that uses an Explorer-style display. The following procedure will help you start any SDI application.

Try This

1. Open Visual C++ if you haven't done so already.
2. Use the File | New command to display a New dialog. Click the Projects tab so that you see a dialog like the one shown in Figure 3-5.
3. Highlight the MFC AppWizard (exe) icon.
4. Type a name for your program in the Project Name field. The sample program uses the name **SDI**.
5. Click OK. You'll see the MFC AppWizard Step 1 dialog box shown in Figure 3-8.
6. Choose the Single document option and click Next three times. You'll see the MFC AppWizard Step 4 of 6 dialog box shown in Figure 3-15. This is where you'll decide on some of the appearance characteristics for your application. Notice that you can choose between standard and Internet Explorer ReBars on this dialog. In addition, you can determine the number of files to allow on the most recently used (MRU) file list. I normally set this value as high as it will go for applications that require access to a lot of files, such as a text editor. Database applications are unlikely to open more than one file, so, in this case, I set the number low, to save memory and reduce clutter in the registry.

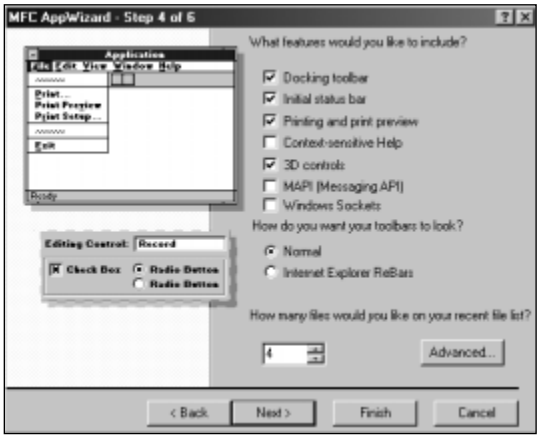

Figure 3-15: SDI applications allow you more flexibility in the appearance of your application, including the use of Internet Explorer ReBars.

7. Choose one or more appearance options. The sample application will use the Internet Explorer ReBars option. All other options will remain at the default setting.
8. Click Next. You'll see a MFC AppWizard Step 5 of 6 dialog box that looks similar to the one shown in Figure 3-10. Notice that you can now choose the style of project you would like to create. The MFC Standard option produces the application style that we're all familiar with — a single blank area for working with a document view. The Windows Explorer option produces an application that contains two panes, just like Windows Explorer does. You can populate these two panes in a variety of ways. For example, a database application could provide a hierarchical view of customer accounts on the left and a single selected record view on the right.
9. Choose a project style, comment inclusion option, and MFC library linking option. The sample application will use the Windows Explorer style and the default setting for the other two options.
10. Click Finish. You'll see a New Project Information dialog like the one shown in Figure 3-16.

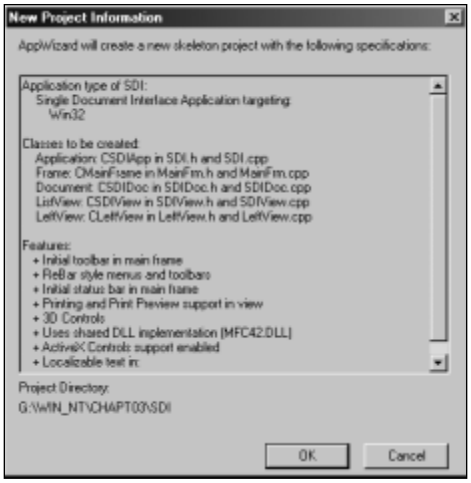
New Project Information
AppWizard will create a new skeleton project with the following specifications:
Application type of SDI:
Single Document Interface Application targeting:
Win32
Classes to be created:
Application: CSDIApp in SDI.h and SDI.cpp
Frame: CMainFrame in MainFrm.h and MainFrm.cpp
Document: CSDIDoc in SDIDoc.h and SDIDoc.cpp
ListView: CSDIView in SDIView.h and SDIView.cpp
LeftView: CLeftView in LeftView.h and LeftView.cpp
Features:
+ Initial toolbar in main frame
+ ReBar style menus and toolbars
+ Initial status bar in main frame
+ Printing and Print Preview support in view
+ 3D Controls
+ Uses shared DLL implementation (MFC42.DLL)
+ ActiveX Controls support enabled
+ Localizable text in:
Project Directory:
G:\WIN_NT\CHAPT03\SDI
OK
Cancel

Figure 3-16: The sample SDI application includes Internet Explorer ReBars and a Windows Explorer style.

11. Click OK. Visual C++ will create an application shell for you.

Displaying information on a Windows Explorer display normally means using some icons to help users differentiate one kind of entry from another. If you were planning on creating some enhanced form of Windows Explorer, you could get those icons directly from Windows or the application you wanted to display. However, a custom application is going to need some icon resource. Our sample application uses two icon resources: One to show when an item is selected, another to show when it's not selected. Figure 3-17 shows the two icons named

IDI_CUST_ICON1 and IDI_CUST_ICON2. The sample application uses an icon size of 32 × 32, but you can use any size that your application might require. For that matter, you don't even have to use icons — you could just as easily use bitmaps. Obviously, the icons you use will need to reflect the use they'll receive in an application. In some cases, you'll need both a selected and deselected icon, in others, a single icon for each type of information you want to present will work just fine.

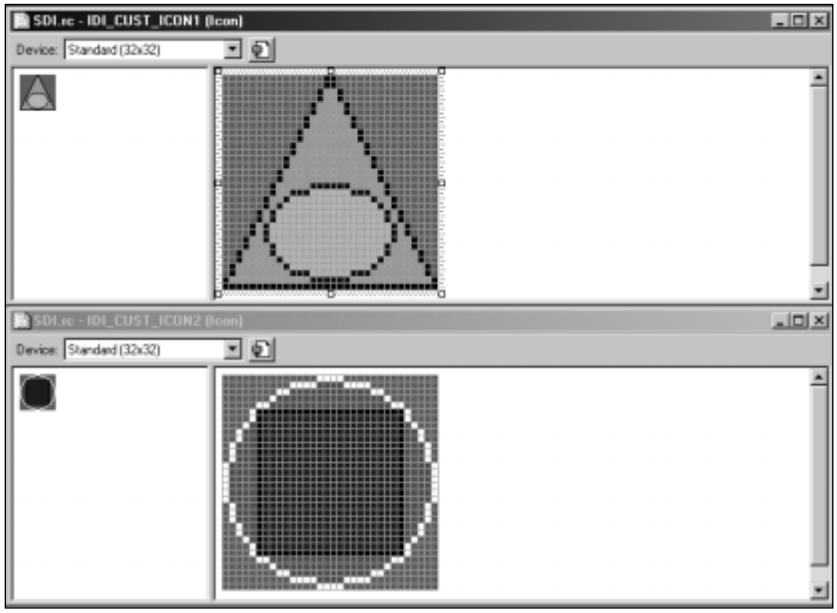

Figure 3-17: Most Windows Explorer(style SDI applications will require some sort of icon support for the items you want to display.

Now that we have an application shell and some icon resources, we can add some code to see how the CTreeView and CListView classes work. We'll actually need to add the code in three stages. The first stage will populate the CTreeView with some entries that we can work with using the OnInitialUpdate( ) method. Whether you can use this method depends on the stability of the left side of your application display. If you're working with something that won't change much like the hierarchical format of your database, then going this route makes sense. On the other hand, if you're displaying a hierarchy that will change during the application session, you'll want to place the code in a separate method and simply call it from the OnInitialUpdate( ) method. Listing 3-3 contains the source needed to populate the CTreeView object (the left side of the Windows Explorer display).

As with the CTreeView, we also need some code for our CListView object. The example places the code in the OnInitialUpdate( ) method for ease of explanation. In a production application, however, you'd need to place this code in a separate

method to allow for the constant updates that this part of the application will experience. In addition, you would need to make the method public if you wanted to link the left and right sides of the display. Our example uses a protected variable for the purpose of tracking the currently selected icon. Listing 3-4 contains the source needed to add icons to the right side of the display. You'll need to include this variable in the protected section of the CSDIView.H file as shown here:

```
protected:
   int   iSelectedItem;   // Selected item tracker.
```

You'll see in a few moments that the selected/deselected feature of CTreeView is automatic. In other words, you don't have to perform any extra coding to automatically change icons when the user selects or deselects an entry in the hierarchy. This feature isn't automatic in the CListView object, however, because it's not something offered by Windows Explorer. In some cases, you may want to change the icon to show state information. With that in mind, the third stage of our project will change the default icon to a selected icon when the user double-clicks on it. Listing 3-5 shows the code you'll need to add for this feature.

Before we can add the code needed to switch between the selected and deselected states on the CListView object, we'll need to add a function shell for performing the task. Use the View | ClassWizard command to display the MFC ClassWizard dialog. Select the Message Maps tab so that we can add a function that will react when the application receives a left-mouse button up message. There are a number of reasons for selecting this particular message that we'll discuss later in the chapter. Choose the CSDIView option in the Class Name field, the CSDIView option in the Object IDs field, and the WM_LBUTTONUP message in the Messages field. Click Add Function. You should see a new function like the one shown in Figure 3-18. Click the Edit Code button to get to the new function so that you can add code to it.

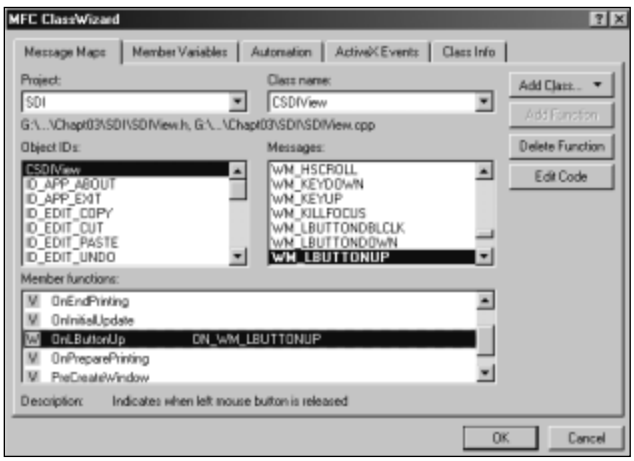

Figure 3-18: The MFC ClassWizard dialog allows you to add functions that respond to specific application messages.

CD-ROM Location: Chap03\SDI
Program Name: SDI.EXE
Source Code Modules in Text: LeftView.CPP

Listing 3-3: **Code for Populating Left Side of Windows Explorer Display**

```
void CLeftView::OnInitialUpdate()
{
   CImageList   *pImageList;     // A list of images to use for display.
   HICON        hIcon;           // The icon handle.
   HTREEITEM    hParent;         // Handle for the parent of a sibling entry.
   HTREEITEM    hSibling;        // Handle for the first sibling level.

   // Default action.
   CTreeView::OnInitialUpdate();

   // Initialize our image list.
   pImageList = new CImageList();
   pImageList->Create(16, 16, ILC_COLORDDB | ILC_MASK, 2, 0);

   // Load some icons.
   hIcon = LoadIcon(AfxGetInstanceHandle(), MAKEINTRESOURCE(IDI_CUST_ICON1));
   pImageList->Add(hIcon);
   hIcon = LoadIcon(AfxGetInstanceHandle(), MAKEINTRESOURCE(IDI_CUST_ICON2));
   pImageList->Add(hIcon);

   // Add the image list to the tree control.
   GetTreeCtrl().SetImageList(pImageList, TVSIL_NORMAL);

   // Add some items to the tree control.
   hParent = GetTreeCtrl().InsertItem("Item 1", 0, 1);
   hSibling = GetTreeCtrl().InsertItem("Item 1.1", 0, 1, hParent);
   GetTreeCtrl().InsertItem("Item 1.1.1", 0, 1, hSibling);
   GetTreeCtrl().InsertItem("Item 1.1.2", 0, 1, hSibling);
   hSibling = GetTreeCtrl().InsertItem("Item 1.2", 0, 1, hParent);
   GetTreeCtrl().InsertItem("Item 1.2.1", 0, 1, hSibling);
   GetTreeCtrl().InsertItem("Item 1.2.2", 0, 1, hSibling);
   hParent = GetTreeCtrl().InsertItem("Item 2", 0, 1);
   GetTreeCtrl().InsertItem("Item 2.1", 0, 1, hParent);
   GetTreeCtrl().InsertItem("Item 2.2", 0, 1, hParent);
}
```

There are a lot of different ways to load the icons we need into the ImageList control. The easiest way is to use an HICON structure, so that's the technique we'll use here. Notice that loading the icon requires an instance handle, which is returned by the call to AfxGetInstanceHandle( ) — again, there are other ways to obtain the handle, but this is probably the easiest if you're using MFC. The

MAKEINTRESOURCE( ) macro allows us to convert the icon resource number into a pointer to its name.

After loading the icons into an ImageList, we need to assign the ImageList to the CTreeView class for our example. Notice, however, that the reference to the tree control is indirect — the -> pointer is implied even though it doesn't appear in the code. The GetTreeCtrl( ) method retrieves the tree control contained in the CTreeView class.

The remaining code in this section shows you how to create a relationship between items in the tree view. The InsertItem( ) method can take a variable number of entries depending on what you want to accomplish. Normally, you'll provide the first three entries which consist of the item name, a selected icon, and a deselected icon. These icon numbers are the numbers assigned to the entries in our ImageList control. A fourth entry in the InsertItem( ) method enables us to create a hierarchy of entries. In this case, there are three levels. The top level requires no handle reference since it's attached to the tree root. The second level is attached to these first-level entries through the hParent handle returned by the call to InsertItem( ). Finally, the third-level entries are taken care of by the hSibling handle.

CD-ROM Location: Chap03\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDIView.CPP

Listing 3-4: **Code for Populating the Right Side of the Windows Explorer Display**

```
void CSDIView::OnInitialUpdate()
{
   CImageList     *pSImageList;     // A list of small sized images.
   CImageList     *pNImageList;     // A list of normal sized images.
   HICON          hIcon;               // The icon handle.

   // Perform the default action.
   CListView::OnInitialUpdate();

   // Initialize our image lists.
   pNImageList = new CImageList();
   pNImageList->Create(32, 32, ILC_COLORDDB | ILC_MASK, 2, 0);
   pSImageList = new CImageList();
   pSImageList->Create(16, 16, ILC_COLORDDB | ILC_MASK, 2, 0);

   // Load some icons.
   hIcon = LoadIcon(AfxGetInstanceHandle(), MAKEINTRESOURCE(IDI_CUST_ICON1));
   pNImageList->Add(hIcon);
   pSImageList->Add(hIcon);
```

```
    hIcon = LoadIcon(AfxGetInstanceHandle(), MAKEINTRESOURCE(IDI_CUST_ICON2));
    pNImageList->Add(hIcon);
    pSImageList->Add(hIcon);

    // Add the image list to the list control.
    GetListCtrl().SetImageList(pNImageList, LVSIL_NORMAL);
    GetListCtrl().SetImageList(pSImageList, LVSIL_SMALL);

    // Add some columns to the list control.
    GetListCtrl().InsertColumn(1, "Name", LVCFMT_LEFT, 100);
    GetListCtrl().InsertColumn(2, "Type", LVCFMT_LEFT, 100);
    GetListCtrl().InsertColumn(3, "Description", LVCFMT_LEFT, 200);

    // Add some Name items to the list control.
    GetListCtrl().InsertItem(0, "Item 1", 1);
    GetListCtrl().InsertItem(1, "Item 2", 0);
    GetListCtrl().InsertItem(2, "Item 3", 0);
    GetListCtrl().InsertItem(3, "Item 4", 0);

    // Update the value of iSelectedItem.
    iSelectedItem = 0;
}
```

The code required to populate the image list control with items is almost the same as the code we saw in Listing 3-4. However, there are subtle differences that you need to take into account when working with a list control versus a tree control. For one thing, a list control has no hierarchical structure, so you don't need to keep track of the various handles returned by InsertItem( ). Another difference is that there isn't a selected or deselected icon for an item. Microsoft assumes that one icon will serve and that icon is probably the one used by an application. However, we can change that behavior using the technique shown in Listing 3-6.

List controls provide a feature not found with the tree control, the ability to present four views of the icons: Large Icons, Small Icons, List, and Details. This feature presents a few challenges for the programmer. For one thing, we need two ImageList controls; one for large icons and a second for small icons. This is the reason for the two image list pointers (pNImageList and pSImageList). Assigning the image lists to the list control is no different than with a tree control — you simply have to tell the list control which kind of image list you want to add.

Creating the Details view is a little more difficult. At a bare minimum, you must provide a single column heading for the item name. Our example shows how to add three column headings. They only appear when you click the Details button on the application toolbar.

CD-ROM Location: Chap03\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDIView.CPP

Listing 3-5: **Code for Changing the Selected Icon in a CListView**

```
void CSDIView::OnLButtonUp(UINT nFlags, CPoint point)
{
   LVITEM          pItem;     // Structure of item attributes.
   POSITION     pos;     // Position of the selected item.

   // Get the currently select item attributes.
   pItem.iItem = iSelectedItem;
   pItem.iSubItem = 0;
   GetListCtrl().GetItem(&pItem);

   // Set the new image value.
   pItem.iImage = 0;
   GetListCtrl().SetItem(&pItem);

   // Get the newly selected item.
   pos = GetListCtrl().GetFirstSelectedItemPosition();
   iSelectedItem = GetListCtrl().GetNextSelectedItem(pos);

   // Set the new image value to selected.
   pItem.iItem = iSelectedItem;
   pItem.iSubItem = 0;
   GetListCtrl().GetItem(&pItem);
   pItem.iImage = 1;
   GetListCtrl().SetItem(&pItem);

   // Redraw the items to reflect changes.
   GetListCtrl().RedrawItems(0, 3);

   CListView::OnLButtonUp(nFlags, point);
}
```

Listing 3-5 shows some specialty code that I added to the OnLButtonUp( ) method. It turns out that you can't easily use a single click to show a selected item in a list view. The problem is that if you attach code to the OnLButtonDown( ) method, the event fires before the user has actually selected the new item, which means that there's no way for you to change the item icon to show its selected state. This same problem will occur for double-click methods. On the other hand, it appears that due to a bug with Visual C++, the OnLButtonUp( ) method only gets fired after a double-click. So

even though the user has selected the item by the time the mouse button is released the first time, it takes a double-click to actually implement the icon change.

The code for changing the item icon is relatively straightforward. You begin by filling a list view item (LVITEM) structure with the current item number. Always set the iSubItem entry to 0 so that Visual C++ knows that you're talking about an item and not some descriptive element. The GetItem( ) method fills the LVITEM structure with the rest of the values required to change the icon. Since we have only two icons in our ImageList, it's easy to switch to the deselected icon by simply changing the value of iImage. Once we make the icon change in our LVITEM structure, it's time to use the SetItem( ) method to change it onscreen.

Getting the number for the newly selected item requires use of the GetFirstSelectedItemPosition( ) method. Unfortunately, this method returns a POS value and not an actual position number. We can convert it to a position number using the GetNextSelectedItem( ) method, which actually returns the first selected item number the first time you call it. Notice that setting the newly selected item icon to its new state follows the same route as the deselected item that I discussed in the previous paragraph.

At this point, our two icons, the one for the selected item and the one for the deselected item, are changed. However, that change won't show up onscreen unless you perform one more step. You have to refresh the icon display using the RedrawItems( ) method. Our example program refreshes all of the icons for the sake of simplicity. However, in a production application containing hundreds of icons in the list view, you may want to limit the refresh to just the icons that have changed by specifying a 0-based start and end icon in the RedrawItems( ) method.

Once you have all of the required code additions in place, you can compile and run your application. As previously mentioned, this example is only meant as an introduction to using the Windows Explorer view. However, it does provide the basis you'll need for creating more advanced examples of your own. Figure 3-3 shows the output from our example program.

A Simple MDI Application

The previous section of the chapter showed you one of the newest alternative application types that you can create with Visual C++. This flexibility enables you to create user friendly applications with a new look that don't require a lot of coding. However, there's still a need for the older application views that we've all used in applications like word processors and graphics editors. That's what this section of the chapter will concentrate on: A simple text-processing application that allows you to open more than one document at a time. The following procedure will help you create the application shell for any standard MDI application.

Try This

1. Open Visual C++ if you haven't done so already.
2. Use the File | New command to display a New dialog. Click the Projects tab so that you see a dialog like the one shown in Figure 3-5.
3. Highlight the MFC AppWizard (exe) icon.
4. Type a name for your program in the Project Name field. The sample program uses the name MDI.
5. Click OK. You'll see the MFC AppWizard Step 1 dialog box shown in Figure 3-8.
6. Choose the Multiple documents option and click Next twice. You'll see the MFC AppWizard Step 3 of 6 dialog box shown in Figure 3-19. See the OLE Feature Settings for MDI and SDI Applications sidebar for a description of the entries on this dialog.

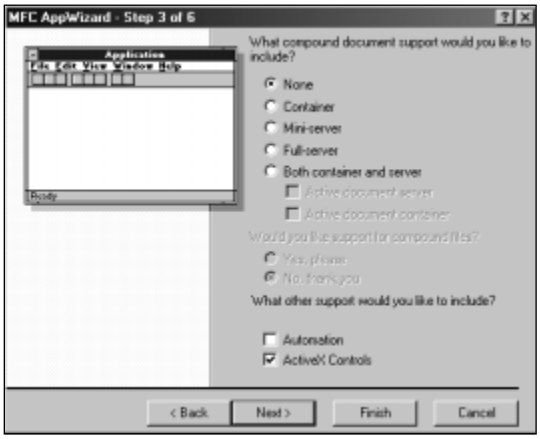

Figure 3-19: Most word-processing applications provide compound-document and full automation support.

7. Choose one or more OLE features for your application. The example application uses the Both Container and Server, Active Document Container, and the Active Document Server options. In addition, make sure you check both the ActiveX Controls and the Automation checkboxes for the example application.
8. Click Next. You'll see the MFC AppWizard Step 4 of 6 dialog box shown in Figure 3-15. Since this is a word-processing application, you'll normally set the most recently used file setting relatively high. The example application uses a setting of 16 since that's the highest setting supported by the current version of Visual C++.
9. Choose between the various application features. There's an extra step we need to take, at this point, since we checked the Automatic checkbox in Step 7.

We have to assign a file extension to our application. Even if we hadn't chosen the Automation option, word processors and other document-handling applications normally have one or more file extensions assigned to them. (Read the sidebar entitled "Using File Extension Templates" for more details.)

10. Click Advanced. You'll see the Document Template Strings tab of the Advanced Options dialog shown in Figure 3-20. This is where you assign a file extension to your application. It's also where you configure any information associated with the file extension such as the long document name.

11. Type any file extension data needed by your application. The example application uses the values shown in Table 3-2.

Figure 3-20: The Advanced Options dialog box allows you to change the file extension and window style settings for your application.

Using File Extension Templates

Both SDI and MDI applications can have a file extension allocated to them as part of the configuration process, though it is a far more common feature with MDI applications. You can also assign file extensions later in the programming process, but doing so is more cumbersome than using the automated route we'll explore in this section.

If you do need multiple file extensions, assigning the first (major) file extension using the automated method provides a template you can use to reduce the complexity of adding another file extension to the application.

Obviously, the big question is whether the application actually requires a file extension. For example, database applications normally have one or two database files hard-wired into the application. The user doesn't need access to a general file extension filter in this case.

Table 3-2
File Extension Settings for the MDI Programming Sample

Field Name	*Setting*
File extension	mdipe
File type ID	MDIPE.Document
Main frame caption	MDI Programming Example
Filter name	MDI Programming Example Files (*.MDIPE)
Doc type name	MDIPE
File new name (short name)	MDIPE

Many programmers have gotten the idea that only three letter file extensions are allowed when creating an application. Nothing could be further from the truth. You can use any number of letters. However, keeping the number of letters short does make it easier to find the document files when needed. As a matter of compromise, I normally use four or five letters for file extensions. This allows me to create file extensions that won't conflict with other applications, and yet are short enough to search for on the hard drive.

12. Click Close to close the Advanced Options dialog.
13. Click Next twice. You'll see the MFC AppWizard Step 6 of 6 dialog shown in Figure 3-21. This dialog tells you all about the classes used to create your application. For most applications, you can accept the default class. However, in some applications, the document and view classes (controlled by the Base class field of the dialog) control the methods used to manipulate documents, which in turn control your application's capabilities. Since Microsoft doesn't have any idea of what kind of document you'll eventually create, they default the MFC AppWizard to use a very generic view and document class. However, much more capable classes are available. Since this application provides text-editing capabilities, we'll use the CRichEditView class as our base class.
14. Choose one of the Base class field options. Click Finish. You'll see a New Project Information dialog like the one shown in Figure 3-22.
15. Click OK. Visual C++ will create a program shell for you.

At this point, you'd normally start enhancing this application to perform certain tasks like working with fonts and searching for text. However, you'll find that the application created for you by the MFC AppWizard is quite functional. Figure 3-4 shows some of the capabilities you get from the default application. We'll further enhance this application in Chapter 4.

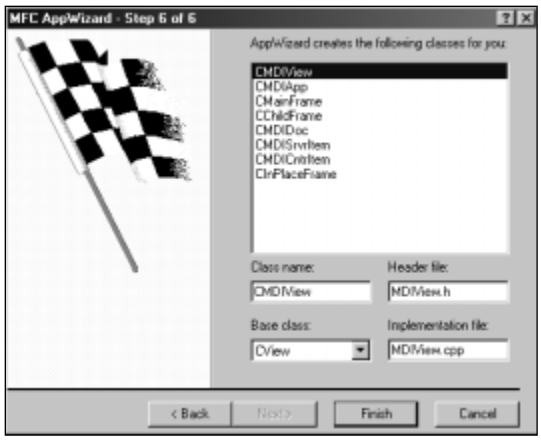

Figure 3-21: The Base class field option you choose can greatly affect the document-handling capabilities of your application.

New Project Information
AppWizard will create a new skeleton project with the following specifications:
Application type of MDI:
Multiple Document Interface Application targeting:
Win32
Classes to be created:
Application: CMDIApp in MDI.h and MDI.cpp
Frame: CMainFrame in MainFrm.h and MainFrm.cpp
MDIChildFrame: CChildFrame in ChildFrm.h and ChildFrm.cpp
Document: CMDIDoc in MDIDoc.h and MDIDoc.cpp
RichEditView: CMDIView in MDIView.h and MDIView.cpp
Container Item: CMDICntrItem in CntrItem.h and CntrItem.cpp
Inplace Frame: CInPlaceFrame in IpFrame.h and IpFrame.cpp
Server Item: CMDISrvrItem in SrvrItem.h and SrvrItem.cpp
Features:
+ Initial toolbar in main frame
+ Initial status bar in main frame
+ Printing and Print Preview support in view
+ 3D Controls
+ Uses shared DLL implementation (MFC42.DLL)
Project Directory:
G:\WIN_NT\CHAPT03\MDI
OK
Cancel

Figure 3-22: Our MDI application will provide advanced text-handling capabilities because it uses CRichEditView as a base class.

OLE Feature Settings for MDI and SDI Applications

Choosing the OLE features you want to add to your application is an important part of the configuration process. Most word-processing applications will provide some level of compound-document support. To a lesser degree, graphics and database applications will provide it as well. Automation is another feature that you'll find in most word-processing and database applications. Only some graphics applications provide this feature because they allow external application manipulation support less often. Almost every application on the market today provides support for ActiveX controls, so adding this feature is a default for any Visual C++ application.

The upper half of the MFC AppWizard Step 3 of 6 dialog determines what level of OLE support you'll add to your application. The more support you add, the larger your application will be. The very basic level of support is to act as a container. A container can act as a client and store linked and embedded objects. The next level of support, Mini-server, allows you to create compound documents. A mini-server can't work standalone. In addition, it can only work with embedded documents. A full server does have the full OLE capability to work as a server, but it can't act as a container. An application like this would work much like Microsoft Paint. You can embed or link a Microsoft Paint document into your application, but Microsoft Paint can't hold objects created by other applications. Finally, the Both Container and Server option gives your application a full array of local OLE support. You can use it as both a server and a client. However, this kind of application won't work with an Internet browser—it's not designed as an ActiveX Document server.

Notice that there are two checkboxes right below the final radio button (Both Container and Server). If you check the first box, Visual C++ will also add the support required to make your application an ActiveX Document Server. An Active Document Server has the ability to create and manage ActiveX documents. The second checkbox, Active Document Container, allows your application to contain ActiveX documents within its frame. In essence, you'll be able to host documents from applications like Microsoft Word or Excel.

There are two other important checkboxes on the MFC AppWizard Step 3 of 6 dialog. The first is Automation. You'll normally want to check this option for larger applications. Automation allows you to manipulate objects created by other applications. It also allows an automation client to manipulate the objects that you expose in your application. The easiest way is to think of automation is as a sort of scripting capability, though that isn't precisely correct.

The second option is ActiveX Controls. You have to check this box if you want to use any OCXs in your application—whether they were designed as ActiveX controls or not.

Summary

This chapter provided you with some insights into what Visual C++ can do for you in the way of desktop applications. We looked at what you can expect from each major application project type and how to implement some simple projects.

- There are six major desktop application project types: Console, Win32, dialog-based, SDI, MDI, and HTML-based.
- Console applications are best for small, non-interactive utilities or for porting older DOS application to Windows.
- Dialog-based applications are normally used for interactive utilities, configuration programs, and informational displays.
- SDI applications are normally used for smaller data-manipulation applications like text editors or drawing programs.
- MDI applications are used for large data-manipulation applications where the ability to compare documents is paramount, like word processors.
- The wizard configuration choices you make greatly affect the program skeleton, and therefore, the end result. Making sure that you start with a good program shell is one way to reduce programming time and enhance application functionality.

✦ ✦ ✦